Analyzing and predicting effort associated with finding & fixing software faults

Maggie Hamill[a], Katerina Goseva-Popstojanova[1b]

[a]*Department of Computer Science and Electrical Engineering, Northern Arizona University, Flagstaff, AZ*
[b]*Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV*

Abstract

Context: Software developers spend a significant amount of time fixing faults. However, not many papers have addressed the actual effort needed to fix software faults.
Objective: The objective of this paper is twofold: (1) analysis of the effort needed to fix software faults and how it was affected by several factors and (2) prediction of the level of fix implementation effort based on the information provided in software change requests.
Method: The work is based on data related to 1200 failures, extracted from the change tracking system of a large NASA mission. The analysis includes descriptive and inferential statistics. Predictions are made using three supervised machine learning algorithms and three sampling techniques aimed at addressing the imbalanced data problem.
Results: Our results show that (1) 83% of the total fix implementation effort was associated with only 20% of failures. (2) Both safety critical failures and post-release failures required three times more effort to fix compared to non-critical and pre-release counterparts, respectively. (3) Failures with fixes spread across multiple components or across multiple types of software artifacts required more effort. The spread across artifacts was more costly than spread across components. (4) Surprisingly, some types of faults associated with later life-cycle activities did not require significant effort. (5) The level of fix implementation effort was predicted with 73% overall accuracy using the original, imbalanced data. Using oversampling techniques improved the overall accuracy up to 77%. More importantly, oversampling significantly improved the prediction of the high level effort, from 31% to around 85%.
Conclusions: This paper shows the importance of tying software failures to changes made to fix all associated faults, in one or more software components and/or in one or more software artifacts, and the benefit of studying how the spread of faults and other factors affect the fix implementation effort.

Keywords: software faults and failures, software fix implementation effort, case study, analysis, prediction.

1. Introduction

The cost of software faults is very high, not just because finding and fixing faults increases the development and testing cost, but also because of the consequences of field failures due to these faults. According to a report from Cambridge University, software developers spend on average 50% of their time finding and fixing bugs, which leads to an estimated cost to the global economy of $312 billion per year [1]. Similarly, Griss found that between 60 and 80 percent of the total development cost was spent on maintenance and rework [2], while Boehm and Basili claimed that software projects spend 40-50% of their effort on avoidable rework [3]. This paper is focused on analyzing factors that affect the effort needed to fix software faults and the ability to predict it using the information provided in software change requests (SCRs). Better understanding of software fixes associated with high effort and factors that affect them support more cost-efficient software debugging and maintenance. Furthermore, predicting the levels of software fix implementation effort is useful for planning and proactively adjusting resources in long-lived software systems that require sustained engineering.

[1]Corresponding author. E-mail: Katerina.Goseva@mail.wvu.edu. Phone: + 1 304 293 9691. Postal address: Lane Department of Computer Science and Electrical Engineering, West Virginia University, PO Box 6109, Morgantown, WV, 26506, USA

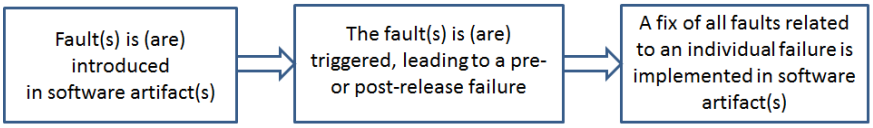

Figure 1: The cause-effect relationships among faults, failures and fixes

Before continuing, we provide definitions of the terms used in this paper, which were adapted from the ISO/ IEC/ IEEE 24765 standard [4]. A *failure* is the inability of a system or component to perform its required functions within specified requirements. A *fault* is an accidental condition or event, which if encountered, may cause the system or component to fail to perform as required. Although every failure is caused by one or more faults, every fault does not necessarily lead to a failure because the condition(s) under which the fault(s) would result in a failure may never be met. To prevent a specific failure from (re)occurring all faults associated with that failure must be corrected. Therefore, we define an additional term, *fix*, which refers to all changes made to correct the fault(s) that caused an individual failure [5]. With respect to definitions appearing elsewhere, the most similar to the term fix (as it is used in this paper) appears to be the term repair, which is defined in the ISO/IEC/IEEE 24765 standard [4] as the correction of defects that have resulted from errors in external design, internal design, or code. Figure 1 illustrates the relationships among faults, failures, and fixes. Note that we considered both observed failures that occurred during operation and potential failures that were prevented from happening by detecting and fixing faults during development and testing.

Understanding the inner-workings of the complex relationships among faults, failures and fixes, as well as the effort associated with investigating failures and fixing faults, are important for cost-efficient improvement of software quality. However, while a significant amount of empirical studies have been focused on studying software faults and/or failures, very few works have tied the faults to the failures they caused and studied the relationships among faults, failures, and fixes. One of the main reasons for this is the difficulty to extract relevant information. Thus, even though most software development projects use some sort of bug, problem or issue tracking system, these systems often do not record how, where, and by whom the problem in question was fixed [6]. Information about fixes is typically hidden in the version control system that records changes made to the source code. In most cases, it is not identified if a change was made because of fixing fault(s), enhancement(s), or implementation of new feature(s). Furthermore, not many projects keep track of the effort needed to investigate failures and fix the corresponding faults.

This paper is part of a larger research effort aimed at characterizing and quantifying relationships among faults, failures and fixes, using the data extracted from a large safety-critical NASA mission. The fact that the mission kept detailed records on the changes made to fix fault(s) associated with each failure allowed us to close the loop from failures to faults that caused them and changes made to fix the faults. In our previous work we showed that software failures are often associated with faults spread across multiple files [7]. Our results further showed that a significant number of software failures required fixes in multiple software components and/or multiple software artifacts (i.e., 15% and 26%, respectively), and that the combinations of software components that were fixed together were affected by the software architecture [5]. More recently, we studied types of faults that caused software failures, activities taking place when faults were detected or failures were reported, and severity of failures [8]. Our results showed that components that experienced more failures pre-release were more likely to fail post-release and that the distribution of fault types differed for pre-release and post-release failures. Interestingly, both post-release failures and safety-critical failures were more heavily associated with coding faults than with any other type of faults.

In this paper we systematically explore the effort associated with investigating and reporting the failures (i.e., *investigation effort*), as well as the effort associated with implementing the fix to correct all faults associated with an individual failure (i.e., *fix implementation effort*). We also focus on predicting the fix implementation effort using the data provided in the SCRs, when the failure was reported. Some related works in this area used the term "fault correction effort". We use the term "fix implementation effort" to emphasize that our focus is on the effort needed to fix (i.e., correct) all faults associated with an individual failure, rather than individual software faults. Based on the processes followed by the NASA mission and our discussions with project analysts, it appears that the values of the effort data used in this paper are of high quality. However, it should be noted that, as in any observational study, it was infeasible to postmortem fully verify and validate the accuracy of self-reported data, such as the investigation effort and fix implementation effort.

It should be noted that investigation effort and fix implementation effort were not studied in our previous works. In general, the analysis and prediction of software fix implementation effort are much less explored topics than the

analysis and prediction of software development effort (see [9] and references therein).

The first part of our study is focused on analysis of investigation effort and fix implementation effort and factors that affect them. We start with the research question:

RQ1: How are investigation effort and fix implementation effort distributed across failures?

Then, we study if investigation effort and fix implementation effort are affected by several factors, such as when failures were detected / observed, severity of failures, spread of faults across components and/or across software artifacts, and by types of faults and types of software artifacts fixed. Each of these factors individually is addressed in research questions RQ2-RQ5:

RQ2: Are investigation effort and fix implementation effort associated with post-release failures higher than corresponding efforts associated with pre-release failures?

RQ3: Are investigation effort and fix implementation effort associated with safety-critical failures higher than corresponding efforts associated with non-critical failures?

RQ4: Do failures caused by faults spread across components or across software artifacts require more effort?

RQ5: Is fix implementation effort affected by types of faults that caused the failure and software artifacts being fixed?

We conclude the analysis part with RQ6 focused on the effect of combinations of factors on the fix implementation effort with a goal to uncover important interactions among factors that cannot be observed by one-factor-at-a-time analysis.

RQ6: How do interactions among factors affect the fix implementation effort?

The second part of our study focuses on predicting the level of fix implementation effort, that is, explores the research question:

RQ7: Can the level of fix implementation effort (i.e., high, medium, or low) be predicted using the information entered in the SCRs, when the failure was reported?

The novel aspects of our work are as follows:

- **We consider the total effort for fixing all faults associated with an individual failure and study how the spread of fixes across components and across different types of software artifacts affect the amount of fix implementation effort.** Each reported failure was mapped to one or more faults that caused it and the corresponding changes made to fix these faults, which allowed us to compute the total effort for fixing all faults associated with an individual failure. These faults may be located in one or more software components and/or in one or more software artifacts. While several related works, focused primarily on prediction [10, 11, 12] or both on analysis and prediction [13], used some measure of spread (e.g., deltas, files, or components), it appears that all related works considered only changes made to the source code. Our study includes faults in all types of software artifacts (e.g., requirements documents, design documents, source code, supporting files, etc.), which led to important insights that have practical implications.

- **We conducted thorough analysis of fix implementation effort and how it was affected by several factors**, which in addition to the spread of fixes across components and across software artifacts included pre-release and post-release occurrence, severity, fault types, and types of artifacts fixed. The only related work that, in addition to prediction, included detailed analysis of fault correction effort (i.e., fix implementation effort) [13] considered smaller number of factors (i.e., the number of software components involved in the correction, the complexity of the components, and the testing phase in which faults occurred). Even more, we explored how interactions of multiple factors affected fix implementation effort, which has not been done in related works. This aspect of our work led to findings that could not be uncovered by means of one-factor-at-a-time analysis.

- **We aimed at predicting the level of fix implementation effort as early as possible**, using only the information available at the time when the failure was reported and before the fix was implemented. Predictions were based on using several supervised machine learning methods for multiclass classification and several treatments for imbalanced data. The performance of the combinations of learners and imbalance treatments was assessed using both per class and overall performance metrics.

The main empirical findings of our work include:

- **Investigation effort and fix implementation effort are not uniformly distributed.** The top 20% of failures ranked by investigation effort accounted for 75% of the total investigation effort, and the top 20% of failures ranked by fix implementation effort accounted for 83% of the total fix implementation effort. Interestingly, some failures that required high investigation effort did not require high fix implementation effort, and vice versa.

- **Post-release failures required more effort**. Post-release failures required 1.5 times more investigation effort and 3 times more fix implementation effort that pre-release failures.

- **Safety-critical failures required more effort.** Safety-critical failures required 2 times more investigation effort and 3 times more fix implementation effort than non-critical failures.

- **Fix implementation effort increased with the spread of associated faults across the components, as well as with the spread across multiple types of software artifacts.** Specifically, 16% of failures that led to fixes in multiple components were responsible for close to 39% of the total fix implementation effort. And, although only 28% of failures led to fixes in more than one type of software artifact, together they were responsible for 60% of the total fix implementation effort.

- **The effect of the spread of fixes, however, differed for pre-release and post-release failures.** In particular, the fix implementation effort related to pre-release failures increased with the spread of faults across components and types of artifacts. On the other side, for post-release failures the fix implementation effort was not significantly impacted by the number of components and/or artifacts being fixed.

- **Some types of faults associated with later life-cycle activities did not require significant effort**, especially if only one type of artifact was fixed. Examples included coding faults that led to fixes only in the code, or integration faults that required fixes only in requirements or fixes only in the code.

- **The level of fix implementation effort was predicted with overall accuracy of 73% using the original, imbalanced data. Oversampling techniques improved the overall accuracy up to 77%.** The tree-based learners J48 and PART outperformed the Naive Bayes learner. The imbalance treatments based on oversampling and SMOTE significantly improved the prediction capability for the high effort failures, which also led to increased overall accuracy.

This papers proceeds by discussing related works in section 2 and details of the case study in section 3. The analysis of the investigation and fix implementation effort is presented in section 4, and prediction of the fix implementation effort is given in section 5. Threats to validity are discussed in section 6 and the conclusion is presented in section 7. Comments on statistical tests and procedures are given in the Appendix.

2. Related Work

The discussion of related works addresses the papers that were focused on analysis and prediction of the time and/or effort associated with implementing fixes (i.e., correcting faults) [10, 11, 14, 15, 16, 12, 17, 18, 19, 13, 20]. Note that neither the prediction of development effort nor the prediction of fault and failure proneness are of interest and therefore are not discussed here; such references can be found in [9] and [21], respectively.

Siy and Mockus studied how domain engineering impacted software change cost, with respect to time spent changing source code [10]. Since the effort data were not available, they were inferred using a regression model based on information from the change management system. Note that this approach of change effort estimation can be used

only after the actual change work has been completed. The results showed that changes made to fix faults were more difficult to make than additions of new code with comparable size.

Mockus et al. used a simple model to predict fault correction effort and its distribution over time [11]. Due to difficulty obtaining actual hours spent making changes, effort values were estimated based on the number of calendar days the Modification Requests (MR) was open. Features considered included the developer who submitted the MR, the type of MR (i.e., new feature, pre-release repairs, field problems), status, MR size, and number of deltas constituting the MR. Based on the predicted effort values it was concluded that field repair MRs were about twice as difficult than a comparable non-field repair MRs.

Zeng and Rine focused on predicting the numerical value of fix effort using the KC1 dataset from the Metrics Data Program [14]. Severity, the mode the system was operating in, type of fault, and number of lines of code changed/added were used as input variables to estimate the fix implementation effort, using dissimilarity matrices and self organizing neural networks. The performance was good for similar projects, but poorer for projects with different development environments.

Paner used machine learning to predict the time to fix a bug, measured in number of days to resolution, discretized into seven equally sized bins [15]. This work was based on Eclipse data extracted from Bugzilla, and did not discuss the attributes used for prediction. The accuracy ranged from 31% to 35%, depending on the algorithm used.

Weiss et al. predicted the numerical values of the fix effort for software issues using Nearest Neighbor approach [16]. The results showed that the prediction accuracy varied greatly for different issue types (e.g., bugs, feature requests, tasks). This work was based on 567 issue reports from JBoss.

Ahsan et al. applied a heuristic based on the number of calendar days bug reports were open and the number of bugs a specific developer worked on during those days to estimate the effort associated with fixes for an open source software application [12]. Even though the spread of the fix across source files/packages was included, it was not discussed. The performance was measured using the Pearson's correlation coefficient between the actual and predicted effort values (which ranged from 51% to 56%), as well as the mean magnitude of relative error (which ranged from 63.8% to 93.6%).

Giger et al. [17] explored the possibility of predicting which bugs are fixed quickly and which are fixed slowly based on threshold values defined by the median number of calendar days bug reports remained open. For the six open source systems considered, accuracy values ranged from 65% to 74%, and improved slightly as more post-release data were included. Although pre- and post-release data were available, it was not studied how effort values differ for bugs reported pre-release vs. post-release.

Abdelmoez et al. used Naive Bayes classifier to distinguish between bugs that were very fast to fix and very slow to fix, using seventeen attributes extracted from Bugzilla's bug reports for four open source systems [18]. The thresholds for bugs that were very fast and very slow to fix were based on quartile split of the number of calendar days bug reports were open (i.e., the first quartile contained bugs that were very fast to fix and the last quartile contained bugs that were very slow to fix). The precision ranged from 39% to 76% for predicting the very fast bugs and from 41% to 49% for predicting the very slow bugs to fix.

Zhang el at. also used the number of calendar days the bug report was open as an effort measure [19]. Bug reports were classified as above or below threshold values using k-Nearest Neighbors with input features such as severity, priority, submitter, and owner. The proposed method was able to predict whether a bug will be fixed slowly or quickly, with F-measure from 65% to 79%.

The closest to the work presented in this paper are the works by Evanco [13] and Song et al. [20]. Evanco's work was focused both on analysis and prediction of fault correction effort using dataset of 509 faults detected during unit testing and system/acceptance testing for three Ada projects [13]. The attributes considered were: number of software components involved in the correction, complexity of components, and testing phase in which faults occurred. The prediction was based on expressing the fault correction effort in terms of a log-linear functional form and calculating probabilities of the effort amount to fall into one of the four categories: less than or equal to one hour, greater than one hour but less than or equal to one work day (8 hours), greater than one day but less than or equal to 3 days, and greater than 3 days. The correlation between predicted and actual classes, for each project, was above 90%. As others, this work did not consider changes in other artifacts besides source code.

Song et al. focused on comparing a prediction model based on association rule mining with other classification methods (i.e., NaiveBayes, C4.5, and PART) [20]. The effort data was classified into the same four categories as [13]. The work was based on NASA's Software Engineering Laboratory issue data consisting of more than 200 projects

and included the following features: fault types (all related to source code only), a flag indicating whether the fault was due to a typographical error, whether code was left out, and/or whether it was result of an incorrect executable statement. The accuracy values of the association mining rule approach (i.e., 94% predicting isolation effort and 95% predicting correction effort) were significantly higher than for other classification models (which ranged from 68 - 72%).

There are several notable differences that distinguish our work from the related works:

- *Data quality and sample size.* We use actual effort values (in hours) entered by analysts and/or developers doing the fault correction work, rather than estimates based on calendar dates [11, 15, 12, 17, 18, 19], effort values inferred from change data [10], or ordinal estimates [13, 20]. Only two related works used numerical effort hours – one was based on a small dataset of only 15 fault fix efforts [14] and in the other it was stated that the authors were warned not to put "too much faith in the effort data" [16]. The work presented here, on the other hand, is based on a large dataset consisting of fixes associated with over 1,200 failures linked to 2,500 faults, with actual effort values for each change entered by analysts or developers. Even though the effort values were self-reported, we believe that the data are of high quality because the work was overseen and values were approved by the supervisors. Furthermore, the data quality, including the reported effort values, were verified through the independent review process followed by the mission. While in an observational study it is infeasible to fully verify the accuracy of the effort data postmortem, the researchers addressed the data quality issues through discussions and informal interviews with the project analysts. Works, such as [19], that used the number of calendar days the bug report remained open as a metric, showed that high severity and priority bug reports had shorter fixing times (i.e., they were open for smaller number of calendar days) than bug reports with lower severity and priority. Our work, which is based on actual effort measured in hours, showed that high priority failures (i.e., post-release failures or safety-critical failures) actually took more time (in number of hours) to fix than low priority failures. Obviously, when a metric based on the number of calendar days a bug report remained open is used, it introduces bias and may produce misleading results because, typically, developers do not work on fixing the bug for the whole duration bug report was open. Measuring effort values based on actual number of hours spent fixing the faults, as in this work, removes this bias and provides a more accurate picture.

- *Detailed analysis of factors that affect fix implementation effort.* We thoroughly analyze fix implementation effort and how it was affected by the following factors: pre- / post-release occurrence, severity, spread of fixes across components and across fixed artifacts, as well as by the interaction of these factors. The works presented in [11, 14, 15, 16, 18] were focused only on prediction; they did not include analysis of the factors and how they affect the effort. Several papers explored effort distribution and concluded that it is skewed [12, 13, 19, 20]. Siy and Mockus [10] discussed the statistically significant predictors in their regression model, which included the number of deltas that, in a way, measures the size and complexity of changes in the source code. This discussion, however, was only in the context of prediction because actual effort values were not available [10]. Zhang et al. [19] indicated that bugs with higher severity and priority were on average fixed faster in terms of number of calendar days. Note, however, that the time to fix a bug in [19] was measured in days the bug report remained open and did not reflect the actual effort in hours spent fixing the bugs. Evanco [13] conducted more detailed analysis than the other related works, but did not consider the pre- and post-release occurrence, severity, and types of software artifacts fixed, all of which are considered in our work. It is important to note that none of the related works considered fixes made to software artifacts other than source code nor did they consider how the interactions of multiple factors affect the fix implementation effort.

- *Prediction of the levels of fix implementation effort.* Most related works on prediction of fault correction effort were focused on predicting the time a bug report remained open [11, 12, 15, 17, 18, 19], which is different metric than the actual effort spent on fixing faults because in most cases developers do not work continuously on fixing a bug during the period the corresponding bug report remains open. Others used models to infer change effort values after the actual work was completed [10]. None of the related works addressed the imbalance problem of effort distribution. Similarly as [13, 20], we predict several levels (i.e., low, medium, and high) of fix implementation effort. We use multiclass classification, based on using three supervised machine learning methods, with features available at the time the SCRs were created and before the changes were made. Unlike

[13, 20] which used only overall metrics to evaluate performance, in addition to overall accuracy, we used per-class metrics (i.e., recall, precision and F-score for each level of effort).

3. Case study and attributes description

In this section we describe the case study following the guidelines suggested in [22, 23] and then provide the description of the attributes used in the analysis.

3.1. Case study description

We start the case study description with the plan of our case study, which includes what exactly was studied, the design of the study, the objective(s), methods for data collection, and selection strategy [22, 24].

We use as a *case* the flight software from a large safety-critical NASA mission. The analyzed data spans almost ten years of development and operation for 21 Computer Software Configurations Items, which represent large-scale software components. Because the mission is treated as a single unit of analysis, the design is considered to be *a single-case holistic design* [25]. The study has several types of *objectives* [22]: exploratory (i.e., seeking new insights and generating ideas for new research), descriptive (i.e., portraying the current status), improving (i.e., aiming to improve certain aspects), and predictive (i.e., predicting level of fix implementation effort). The *data collection* is a crucial part of conducting any case study [25]. The change tracking system of the NASA mission stores Software Change Requests (SCRs) entered by analysts, which are filed at the component level. Our work is focused on SCRs entered when non-conformance to a requirement was observed, which by definition represent failures. Each non-conformance SCR was reviewed by a board and, if it was determined that the problem needs to be fixed, it was assigned to an analyst who recorded data, including the type of fault(s) that caused the failure and severity of the failure. A review board verifies the data recorded in SCRs and approves any solution before it is implemented, which helps ensure that changes made to software artifacts fix the appropriate faults. During the *selection process*, to ensure data quality, we removed SCRs that were missing mandatory fields, were withdrawn, tagged as duplicates, and/or tagged as operator errors.

Next, we describe the *context* of the case study [26, 27], i.e., the product, the development process, as well as the specific practices and techniques followed by mission personnel. The product is an embedded flight software system of an active NASA mission. The overall system is divided into Computer Software Configuration Items (CSCIs), each containing between 81 and 1,368 files. The functionality of these CSCIs spans from command and control; generation, distribution, storage, and management of supporting utilities; and failure detection, isolation, and recovery; to human-computer interfaces and scientific research support. The mission's software system follows a hierarchical architecture consisting of three levels, with a single top level component [5]. The work in this paper is based on 21 main CSCIs, which together have over 8,000 files and millions of lines of code. Software components were built following an iterative development process; each component has its own release schedule. The development teams belonged to several locations. (The number of locations varied from two to four throughout the project lifetime.) Due to the safety-critical nature of the software, development organizations followed traditional software development life cycle phases. The quality assurance process spanned throughout the life cycle, that is, pre-release (i.e., development and verification & validation, including independent verification & validation) and post-release (i.e., on-orbit). It included different software artifacts (e.g., requirements, design, code, procedural issues, etc). The quality assurance activities included inspections, analysis (both manual and using static code analysis tools), and different types of software testing (e.g., dry run testing, formal testing, integration testing, regression testing, acceptance testing). The effectiveness of these activities in detecting faults was analyzed in our previous work [8].

The*main practices followed by the mission personnel* are described next. Fault detection activities, focused on identifying non-conformance to requirements, were conducted and recorded throughout the software development life cycle. Upon observing non-conformance (or potential for non-conformance) with requirements, an analyst creates a non-conformance SCR in the mission's problem reporting and corrective action database. These non-conformance SCRs represent either potential failures (i.e., those revealed pre-release during development and testing) or post-release failures (i.e., those that were observed on-orbit). Each non-conformance SCR identifies the failed component (i.e., where the non-conformance was observed), the release of that component, the root cause of the failure (i.e., fault type), the activity taking place when the fault was detected or the failure was revealed, the severity of the failure and the effort (in hours) required to report the non-conformance.

Changes made to address non-conformance SCRs (i.e., fix the associated fault(s)) are tracked in the form of Change Notices (CNs), which are stored in the same database and linked to the SCR the changes addressed. Each CN is associated with a single component, and identifies the software artifact that was fixed (e.g., requirements, design, code, etc.), as well as the effort (in hours) to implement the changes. In general more than one CN can be linked to a single SCR, which allowed us to consider the spread of the fix among multiple components and types of artifacts. Even though the effort values were self-reported, we believe that the data are of high quality because: (1) effort values were collected at a low level of granularity, in an explicit CN field, as each change was made, (2) the work was overseen and entered values were approved by supervisors, and (3) SCRs and CNs were reviewed by an independent review board. Following these rigorous practices minimizes the chances of inaccurate effort values, unlike the tendency of reporting overestimated and underestimated effort values in cases when the work was done in uncontrolled working environment, with subjects who were not overseen directly [28]. Note, however, that as in any observational study, the accuracy of the data could not be fully verified and validated postmortem.

Last but not least, we would like to emphasize the close collaboration between the research team and the NASA personnel, which included formal and informal interviews. The domain expertise provided by the NASA personnel helped us better understand the data. The results and lessons-learned reports delivered to the sponsoring agency and those published in the research literature (including this paper) were completed through an iterative process with NASA personnel. Multiple cycles of analyzing, presenting, reviewing, and re-analyzing occurred. This, along with the procedures and processes followed by NASA with respect to data collection and review, provides assurance of the quality of collected data and accurate interpretation of the results.

Table 1 shows for each of the 21 components considered in this paper the size in number of files and lines of code (LOC), as well as the numbers of SCRs and CNs associated with it. All non-conformance SCRs that had at least one CN and 'closed' change status were included in the analysis. A total of 1,257 non-conformance SCRs which satisfied these criteria were linked to total 2,620 CNs, out of which 2,496 CNs shown in Table 1 were linked to the 21 components analyzed in this paper. Note that out of the total 1,257 SCRs, both the investigation and fix implementation effort were recorded for 1,153 SCRs (i.e., 91.7%).

3.2. Description of attributes

The analysis presented in this paper is based on the following attributes, which were extracted from the non-conformance SCRs and associated CNs:

- *Fault type* refers to the root cause of the failure. For each non-conformance SCR, analysts select the fault type value from a pre-defined list. With the help of the project personnel these values were grouped into the following major categories: *requirements faults*, *design faults*, *coding faults*, *integration faults*[2], and *other faults*. Similarly as in case of the Orthogonal Defect Classification (ODC) [29], the values of the fault type attribute are mutually exclusive. That is, there is only one root cause of a failure, although it may have multiple manifestations. For example, if a failure occurred during integration testing and it was caused by a coding error, which in turn was due to missing requirement, the fault type would be classified as a requirements fault.

- *Pre- / post-release occurrence* specifies when the failure was exposed. This was determined using the detection activity field in the SCR. Pre-release occurrence is related to detection activities such as inspections, audits, analysis (e.g., informal reviews and walkthroughs), and various types of testing (e.g., dry run testing, formal testing, integration testing, regression testing, acceptance testing). The on-orbit activity represents the cases where failures occurred post-release.

- *Severity* is the potential or actual impact of the failure on the system. Severity is initially recorded in the SCR, but may be updated in the associated CNs. In this paper we use safety-critical, non-critical, and unclassified as severity values.

- *Investigation effort* captures the hours spent investigating the observed failure and filing the associated SCR. The investigation effort value is entered in the SCR. In simple cases it only includes the time spent filing the

[2]Integration faults type in this paper integrates the 'data problems' and 'integration faults' types used in our previous work [7].

Table 1: Distribution of SCRs and CNs across 21 components

Component	# of files	Size in LOC	# of non -conf SCRs	# of CNs
1	207	48,910	179	252
2	200	60,386	8	14
3	287	78,854	4	3
4	228	46,657	4	6
5	269	71,953	4	6
6	321	92,978	15	18
7	289	34,938	33	43
8	270	43,012	44	86
9	25	21,266	1	9
10	356	83,134	8	10
11	444	103,145	24	36
12	277	55,475	11	17
13	599	57,800	35	72
14	280	27,940	28	53
15	84	38,882	23	44
16	169	47,541	44	70
17	587	147,520	112	158
18	552	174,614	104	169
19	747	164,419	102	197
20	1368	737,504	412	1157
21	415	74,618	62	76
Total	8,071	2,211,546	1,257	2,496

SCR, while in other cases it includes additional time spent gathering evidence to be presented to the review board, such as re-running test cases or building data files. Note that investigation effort does not include time spent isolating faults, and thus cannot be compared to the isolation effort used in [20].

- *Number of fixed components* identifies the number of components affected by changes made to fix all faults associated with an individual failure. Since each CN refers to only a single component, the number of fixed components is determined by counting the unique components across all CNs linked to a single SCR.
- *Fixed artifacts* refers to the type of artifacts affected by changes made to fix faults and prevent failures from (re)occurring. Unlike related works, which focused on changes made only in the source code, we analyzed changes made in multiple types of software artifacts, including: *requirements documents*, *design documents*, *code files*, *supporting files*, *tools*, and *notes/waivers*, which typically serve to identify non-conformance SCRs that have not yet been fixed.
- *Fix implementation effort* is the total effort (in hours) spent making changes needed to fix all faults associated with each individual failure. This is the cumulative effort associated with all CNs linked to each non-conformance SCR. The time spent implementing changes is entered in the effort field of each CN by the person making the changes, overseen and approved by the supervisor, and reviewed by the review board. Fix implementation effort is the response variable used for predictions in section 5.

4. Analysis of failure investigation effort and fix implementation effort

Research questions RQ1-RQ6 are focused on the characterization of investigation effort and fix implementation effort and specifically explore how they are affected by different factors (i.e., attributes) defined in section 3.2. Wherever relevant, in addition to descriptive statistics, we report the results of the inferential statistics. More detailed comments on the statistical tests and procedures used in the paper are presented in the Appendix.

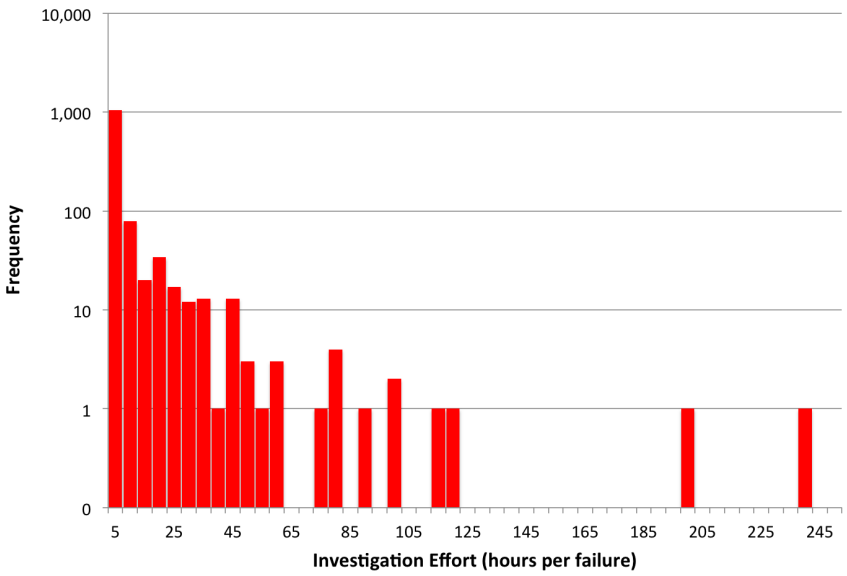

Figure 2: A histogram of investigation effort per individual failure

4.1. RQ1: How are investigation effort and fix implementation effort distributed across failures?

For the 1,257 non-conformance SCRs (i.e., failures) analyzed, a total effort of over 6,700 hours was spent on investigating and reporting failures, across all components and all considered SCRs. Figure 2 shows a histogram of investigation effort per failure. The investigation effort values ranged from 0 to 240 hours, with a median value of 2 hours per failure. For investigation effort, we found evidence of the Pareto principle, that is, **20% of failures accounted for 75% of the total effort spent investigating all failures, which indicates a very skewed distribution**. 44% of failures required one hour or less to investigate, an additional 44% required more than one hour but less than one working day (i.e., 8 hours), and only 12% required more than one working day (i.e., more than 8 hours), which is consistent with results presented in [13].

As described in section 3.2 the fix implementation effort represents the total effort (in hours) spent implementing fixes for all CNs associated with an individual SCR. Of the total 1,257 SCRs, for around 92% (i.e., 1,153 SCRs) the associated CNs included values in the effort field. Cumulatively these 1,153 SCRs took 17,500 hours to fix and, based on the SCR investigation effort field, 2,140 hours to investigate. Figure 3 presents a histogram of fix implementation effort per individual failure. The median value is 3 hours per SCR, with a range from 0 to 1,193 hours. Again, we found evidence of the Pareto principle, that is, **20% of SCRs accounted for 83% of total fix implementation effort**. Thus, although fixing the faults associated with most SCRs required little effort (in the range of several hours), a few SCRs required significantly more effort, in hundreds or even thousands of hours. Interestingly, the top 20% of SCRs ranked by fix implementation effort only accounted for 5.5% of the total investigation effort spent across the 1,153 SCRs that recorded both the investigation effort and implementation effort. We computed the Kendall τ rank correlation coefficient. Its value of 0.4 indicates that SCRs associated with high investigation effort may not always require high fix implementation effort.

Our results are consistent with related work [13] in terms of the nonuniform distribution of effort spent fixing faults. However, a larger percent of failures in our case study were associated with fixes that required more than eight hours effort than in Evanco's study [13] (i.e., 28% in our study compared to 7-15% in [13]). One of the reasons for this may be the fact that we considered fixes to all types of software artifacts, while [13] focused on changes to the source code only. A nonuniform distribution of effort was also reported in [12, 19], which measured the effort in number of calendar days to the bug resolution.

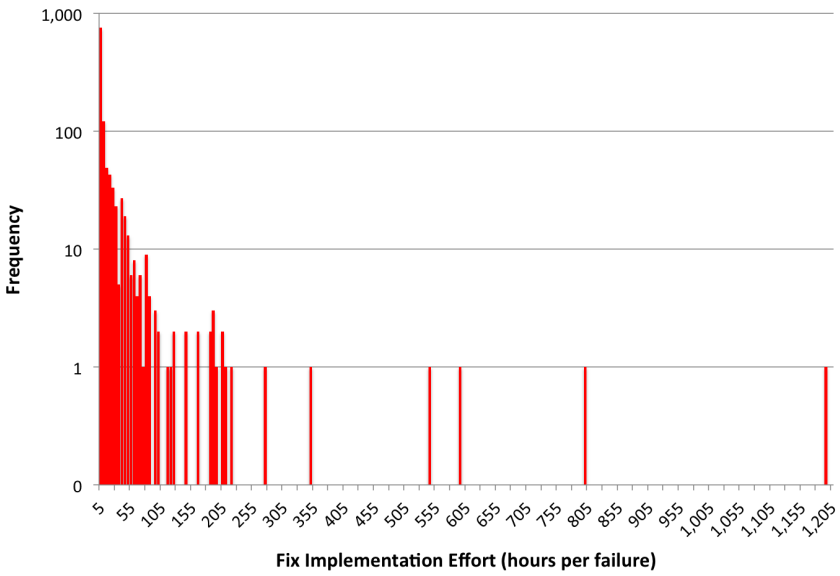

Figure 3: A histogram of fix implementation effort per individual failure

Table 2: Investigation effort (in hours) and fix implementation effort (in hours) for pre- and post-release failures

	# of SCRs	Total effort	Mean effort per SCR	Standard deviation	Median effort per SCR	SIQR
Investigation Effort						
Pre-release	1,116	2,390.0	2.1	2.4	1	0.5
Post-release	37	1,881.9	3.3	4.0	2	1.0
Fix Implementation Effort						
Pre-release	1,116	15,609.9	14.0	43.9	3	4.5
Post-release	37	1,881.9	50.9	195.6	7	12.5

4.2. RQ2: Are investigation effort and fix implementation effort associated with post-release failures (i.e., on-orbit failures) higher than corresponding efforts associated with pre-release failures?

Table 2 shows the mean and standard deviation of investigation effort and fix implementation effort for pre- and post-release failures. It also includes the median and Semi Interquartile Range (SIQR), which is a measure of dispersion around the median and is computed as one half the difference between the 75th percentile and the 25th percentile. Note that the median and SIQR are not affected significantly by outliers, and therefore provide good representation of skewed distributions [30].

From Table 2 it follows that **post-release failures required more investigation effort, as well as more fix implementation effort than pre-release failures**. Specifically, post-release failures on average required approximately one additional hour to investigate, but more than three times the fix implementation effort. The effort amounts for post-release failures also had significantly higher variability measured by the standard deviation and SIQR. For investigation effort, the Mann-Whitney test rejected the null hypotheses that the two populations are the same in favor of the alternative hypotheses that investigation effort associated with post-release failures is higher than for pre-release failures. The result is significant at 0.05, with $p = 0.0118$. Similarly, the Mann-Whitney test also rejected the null hypothesis in favor of the alternative hypotheses that fix implementation effort associated with post-release failures is higher than for pre-release failure ($p = 0.0175$). The significant amount of extra time spent fixing post-release failures is likely due to the facts that (1) more artifacts exist post-release and (2) analysts must test and document post-release (i.e., on-orbit) failures more carefully. Note that one of the related works [11], based on a predictive effort model, showed that the field (i.e., post-release) repair modification records were about twice as difficult than a comparable non-field repair modification records, which is consistent with our result.

Table 3: Investigation effort (in hours) and fix implementation effort (in hours) for safety-critical, non-critical and unclassified failures

	# of SCRs	Total Effort	Mean effort per SCR	Standard Deviation	Median Effort per SCR	SIQR
Investigation Effort						
Safety-critical	122	533	4.4	3.5	3	2.0
Non-critical	650	1,432	2.2	2.6	1	0.5
Unclassified	381	547	1.4	1.1	1	0.5
Fix Implementation Effort						
Safety-critical	122	5,549	45.1	120.5	10.5	22.0
Non-critical	650	9,463	14.6	49.8	4.0	5.0
Unclassified	381	2,479	6.5	13.0	2.0	1.5

4.3. RQ3: Are investigation effort and fix implementation effort associated with safety-critical failures higher than corresponding efforts associated with non-critical failures?

Table 3 shows the mean, standard devition, median, and SIQR of the investigation effort and fix implementation effort for safety-critical, non-critical, and unclassified failures. Based on these results we conclude that **safety-critical failures require more investigation effort, as well as more fix implementation effort than non-critical failures.** Specifically, investigating safety-critical failures took on average twice as long as investigating non-critical failures. Fixing faults associated with safety-critical failures required on average three times the effort of fixing non-critical failures. Fix implementation effort of safety-critical failures also exhibited higher variability. Unclassified failures had the smallest mean and median and the smallest variability among all failures.

To statistically confirm these results, we used the Jonckheere-Terpstra test, which rejected the null hypotheses in favor of the ordered alternative hypotheses that (1) investigation effort of unclassified failures is smaller than investigation effort of non-critical failures, which is smaller than investigation effort critical failures ($p = 2.2 \cdot 10^{-16}$); (2) fix implementation effort of unclassified failures is smaller than fix implementation effort of non-critical failures, which is smaller than fix implementation effort of critical failures ($p = 6.2 \cdot 10^{-15}$).

We suspect that fixing faults associated with safety critical failures required more effort due to the need to test safety-critical failures more carefully and document them better. Although these results may be expected, reporting them is important and significant because, to the best of our knowledge, the extra amount of effort required for safety-critical failures has not been quantified before.

4.4. RQ4: Do failures caused by faults spread across components or software artifacts require more fix implementation effort?

In this subsection, we explore how the fix implementation effort is affected by the number of components fixed and the number of unique types of artifacts fixed. (For example, if both a requirements document and code were fixed with respect to an individual SCR, we count that as two types of software artifacts.) Based on the results shown in Tables 4 and 5, we made the following observations:

- **Fix implementation effort increased with the spread of associated faults across components.** 16% of failures led to fixes in multiple components, but they were responsible for close to 39% of the total effort. Both the mean and median fix implementation effort per SCR increased as the number of fixed components increased from one to four (see Table 4). The Jonckheere-Terpstra test rejected the null hypothesis in favor of the ordered alternative hypotheses that fix implementation effort of failures associated with fixes in one component is smaller than when fixes are spread in two components, which is smaller than when fixes are spread across three and four components ($p = 2.2 \cdot 10^{-16}$). (Due to the small sample size, the two SCRs that required fixes in five components were not included in the analysis, but are given in Table 4 for completeness.)

- **Fix implementation effort increased with the spread of faults across multiple software artifacts.** Only 28% of failures led to fixes in more than one type of artifact, but they were responsible for 60% of the total fix implementation effort. The two artifacts most commonly fixed together were requirements and code. As shown in Table 5, the mean and median fix implementation effort per SCR increased significantly as the number of types of artifacts increased from one to three. Again, the Jonckheere-Terpstra test rejected the null hypothesis,

Table 4: Fix implementation effort (in hours) per number of components fixed

# of components fixed	# of SCRs	Total Effort	Mean Effort per SCR	Standard Deviation	Median Effort per SCR	SIQR
1	968	10,730.9	11.1	32.7	2.0	1.8
2	155	5,035.3	32.5	125.1	10.3	5.8
3	23	1,406.6	61.2	29.0	17.0	3.0
4	5	304.0	60.8	0.8	53.1	0.3
5	2	15.0	7.5	5.0	7.5	0.9

Table 5: Fix implementation effort (in hours) per number of unique types of artifacts fixed

# of types artifacts fixed	# of SCRs	Total Effort	Mean Effort per SCR	Standard Deviation	Median Effort per SCR	SIQR
1	830	6,924.3	8.3	43.8	2.0	2.3
2	259	6,329.5	24.4	61.1	6.0	8.8
3	61	3,427.3	56.2	63.4	36.5	25.6
4	3	810.8	270.3	458.3	10.5	199.6

showing that fix implementation effort of failures associated with fixes in one type of artifact is smaller than fix implementation effort of failures associated with fixes in two types of artifacts, which is smaller than the fix implementation effort of failures with fixes spread across three types of artifacts ($p = 2.2 \cdot 10^{-16}$). (Due to the small sample size, the three SCRs that led to fixing four different types of artifacts were not included in the discussion, but are provided in Table 5 for completeness.)

These results clearly illustrate the importance of associating software failures with all corresponding faults that led to these failures and exploring how their spread across components or across software artifacts affect fix implementation effort. It should be noted that the effort prediction model presented in [14] considered the size of changes in number of source code lines changed or added, but it did not account for the spread of these changes. Other works considered the size of changes in terms of number of deltas that can be in one or more files [10, 11] or number of source code files that were changed to fix the faults [12], but were mainly focused on effort prediction and lacked detailed analysis on how effort values were affected by the spread. Only one of the related works which considered the spread of faults (in this case across multiple components), in addition to the prediction model, presented detailed analysis [13]. As noted previously, it appears that the spread across multiple software artifacts was not considered previously in any of the related works.

4.5. RQ5: Is fix implementation effort affected by types of faults that caused the failure and the software artifacts being fixed?

Our previous work showed there is a statistically significant association between fault types and groups of fixed artifacts [5]. Here, we investigate how fix implementation effort is distributed across these two dimensions. For example, we explore the effort to fix requirements faults that resulted in changes to requirements documents and compare it to the effort needed to fix requirements faults that resulted in changes to requirements documents and source code.

Table 6 and Figure 4 show the mean fix implementation effort for the combinations of fault types and groups of fixed artifacts. The 'Other' category includes groupings of artifacts that were not commonly changed, such as supporting files only, or 'code and tools'. Based on the results in Table 6 and Figure 4 we made the following observations:

- **Requirement faults caught early typically required less effort to fix**. Requirements faults that were corrected by fixing only requirements documents required on average 3.5 hours per SCR, which is much less than fixing requirements faults that led to fixes in multiple types of artifacts. These ranged from on average 17.7 hours per SCR when both requirements documents and code were fixed to 83.3 hours per SCR when requirements, code and supporting files were fixed.

Table 6: Mean fix implementation effort (in hours) per individual failure for combinations of fault types and groups of fixed software artifacts. The numbers in parentheses represent the numbers of instances.

Fault Type	Groups of types of artifacts fixed together						
	Requirements	Requirements, Design & Code	Requirements & Code	Requirements, Code & Supporting Files	Code	Code & Supporting Files	Other
Requirements	(249) 3.5	(8) 45.0	(87) 17.7	(16) 83.3	(32) 39.8	(6) 53.4	(19) 9.3
Design	(4) 2.5	(2) 140.0	(9) 15.1	(4) 21.8	(25) 11.4	(8) 5.2	(9) 2.5
Code	(27) 6.9	(1) 10.4	(56) 40.2	(8) 29.1	(216) 10.0	(30) 30.7	(20) 19.4
Integration	(199) 8.7	(2) 15.5	(23) 16.8	(9) 86.2	(19) 3.8	(14) 30.1	(6) 135.2
Other	(9) 6.5	(0) 0.0	(8) 16.6	(1) 17.0	(22) 4.8	(1) 32.0	(4) 11.4

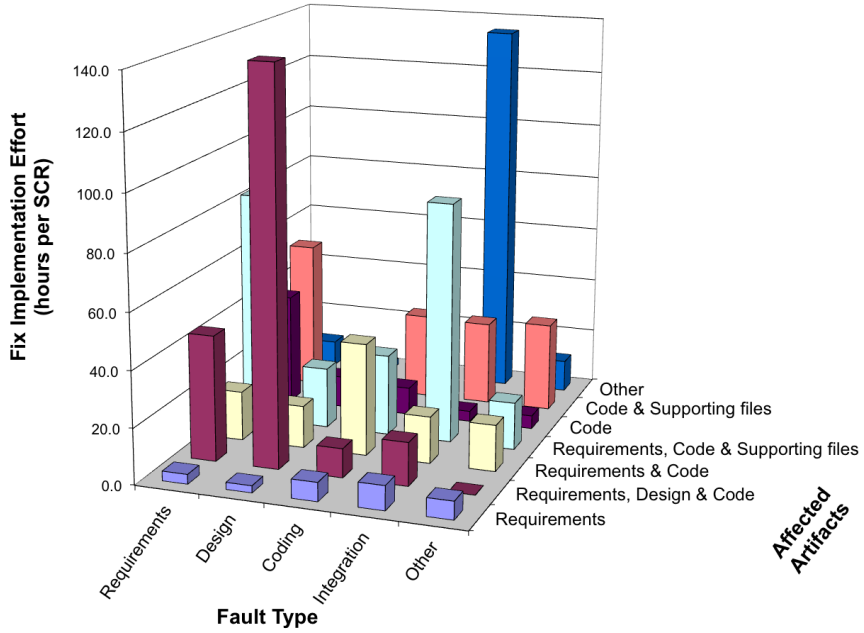

Figure 4: Mean fix implementation effort (in hours) per individual failure for combinations of fault types and groups of fixed artifacts

- **Coding faults that led to changes only in code or only requirements required relatively low effort to fix**, on average 10 and close to 7 hours, respectively. Coding faults that led to fixes in multiple software artifacts required from 20 to 40 hours per SCR to fix. (Coding faults that led to changes in design, requirements and code needed 10.4 hours per SCR, but only one such SCR existed.) The most interesting finding is that over 60% of coding faults (i.e., 216 instances) resulted in fixes only in the code and did not require significant effort.

- **Some categories required more effort to fix than others.** For example, SCRs due to requirements faults, as well as SCRs due to integration faults that led to changes in requirements, code and supporting files required relatively high effort to fix (i.e., over 83 and 86 hours per SCR, respectively). The combinations with the fix implementation effort greater than 100 hours per SCR are all based on relatively small number of instances, which is typical for skewed distributions that have large value observations with small, but non-negligible probability.

- **Types of faults associated with later life-cycle did not necessarily require more effort, especially if only one type of artifact was fixed.** For example, coding faults that led to fixes only in code required 10 hours on average to fix. Integration faults that required fixes only in requirements needed only 8.7 hours to fix[3].

[3]Integration faults that were fixed by changes to requirements were traced to changes in pre-defined look-up tables that were created during the requirements phase.

Table 7: Fix implementation effort values (in hours) for multifactorial analysis

Bin	Safety-critical failures	Post-release failures	More than one artifact type fixed	More than one comp. fixed	Number of instances in the bin	Fix implementation effort			
						Mean	Median	Min	Max
1	0	0	0	0	684	5.83	2.00	0.00	220.00
2	0	0	0	1	93	12.23	8.10	0.20	83.50
3	0	0	1	0	163	21.53	5.00	0.00	594.00
4	0	0	1	1	69	43.62	17.00	1.00	799.40
5	0	1	0	0	14	14.11	3.00	0.50	80.00
6	0	1	0	1	1	8.00	8.00	8.00	8.00
7	0	1	1	0	4	15.85	10.85	0.20	41.50
8	0	1	1	1	3	10.53	5.00	0.60	26.00
9	1	0	0	0	26	4.88	1.00	0.00	30.00
10	1	0	0	1	5	25.26	11.00	1.00	68.20
11	1	0	1	0	61	40.06	14.10	0.00	350.00
12	1	0	1	1	15	84.79	58.00	4.00	271.00
13	1	1	0	0	5	29.84	16.00	0.20	80.00
14	1	1	0	1	2	597.50	597.50	2.00	1193.00
15	1	1	1	0	7	32.46	4.00	0.20	161.90
16	1	1	1	1	1	10.00	10.00	10.00	10.00

Integration faults that were fixed only by changes into the code required even less effort (3.8 hours). This seems to be one of the most interesting findings because it contradicts the anecdotal belief that issues that surface later in the life-cycle are more expensive to fix. Obviously, for significant number of late life-cycle faults the fix implementation effort was not significant if fixes were limited to only one type of artifact.

It should be noted that the effects of the combinations of fault types and groups of fixed software artifacts on the fix implementation effort have not been studied in the related works. Our results based on this analysis confirmed some anecdotal knowledge, but also provided new, interesting findings that have not been reported previously.

4.6. RQ6: How do interactions among factors affect the fix implementation effort?

By definition, two or more factors interact when the effect of one factor on the response variable (in our case the fix implementation effort) depends on the levels of other factor(s). Multifactorial analysis holds potential to uncover patterns due to factor interactions that cannot be observed by means of one-factor-at-a-time analysis.

To carry on multifactorial analysis, we first discretized each factor into two levels, that is, non-critical failures and safety-critical failures, pre-relase and post-release failures, one type of software artifact and more than one type of software artifacts being fixed, and one component and more than one component being fixed (in Table 7 these pairs of levels are annotated with 0 and 1, respectively). Then, we grouped failures into 16 possible bins representing all possible combinations of four factors with two levels each. Table 7 shows the mean, median, minimum and maximum fix implementation effort values for each bin.

Based on Table 7 we made the following observations:

- **Spread of fixes across artifacts is more costly than the spread across components**. As shown in section 4.4, fixes that involve multiple components and/or multiple artifacts require more effort. Looking at the combinations of multiple factors allowed us to compare the fixes that involve a single type of artifact across multiple components (i.e., bins 2, 6, 10, 14) with the fixes that involve multiple artifacts associated with a single component (i.e., bins 3, 7, 11,15). By comparing bin 2 to 3, bin 6 to 7, and bin 10 to 11, respectively, it can be seen that the spread of faults across artifacts required larger fix implementation effort then the spread of faults across multiple components. The only exception from this observation are bins 14 and 15, which is due to one outlier in bin 14 that resulted in 1,193 hours of fix implementation effort.

- **For pre-release failures, both non-critical and safety-critical, fix implementation effort shows a clear increasing trend with the spread of faults across components and types of artifacts**, which is evident from

fix implementation effort values for non-critical pre-release failures (bins 1, 2, 3, 4) and for safety-critical pre-release failures (bins 9, 10, 11, 12). Specifically, faults localized in one component and one type of artifact required the least effort to fix, followed by faults in more than one component and a single type of artifact, then followed by faults localized in one component but spanning more than one type of artifacts. Faults spread across both multiple components and multiple artifacts were the most costly to fix.

- **The effort spent fixing post-release failures, both non-critical and safety-critical, is not significantly impacted by the number of components and/or artifacts fixed.** This is likely due to the fact that more detailed analysis and testing was conducted for all post-release failures, which means additional components and/or artifacts may have been examined even though changes were not required in all locations. It is important to note that this observation is based on bins that have small sample sizes (i.e., bins 5-8 and 13-16). Again, due to one outlier, the only exception was bin 14.

These results highlight the importance of carrying on multifactorial analysis and accounting for the effect of factor interactions on fix implementation effort, which is done for the first time in this paper. For example, it seems counterintuitive that the spread of faults across artifacts leads to more costly fixes than the spread of faults across components. Furthermore, it is somewhat surprising that the effort fixing the faults associated with post-release failures (both non-critical and safety-critical) is not affected significantly by the spread of faults across artifacts and components. None of these observations could have been made by analyzing each factor in isolation.

5. Prediction of the fix implementation effort

In this section we focus on RQ7, that is, on predicting the level of fix implementation effort from the information provided in SCRs, before fixes have been implemented. Specifically, we explore whether supervised machine learning methods can be used to classify the three levels (i.e., classes) of fix implementation effort: high, medium, and low.

5.1. Feature extraction and class definition

Classification of the level of fix implementation effort is based on the features given in Table 8. The first six features are extracted from SCRs, while the last two are based on knowledge about the software architecture.

Numeric prediction of the fix implementation effort would not have been accurate because all features, with exception of Severity and Investigation effort, are categorical. Instead, we discretized the fix implementation effort to three levels (i.e., low, medium, and high) and used multiclass classification for prediction. Specifically, the fix implementation effort values were first transformed by taking the log and then dividing the range of the logs into three levels (i.e., three classes). This is a standard approach for discretization of skewed data, which has been used for development effort prediction [31]. Following this approach, fix implementation effort values smaller than and equal to 1.44 hours were considered low fix implementation effort, those between 1.45 hours and 41.44 hours were considered as medium, and those with values equal to or higher than 41.45 hours were considered high fix implementation effort. The number of instances in the low, medium, and high effort levels were 399, 661 and 93, respectively. This imbalanced distribution was expected considering that 20% of SCRs accounted for 83% of the total hours of fix implementation effort, as reported in section 4.1. The 8% of SCRs classified as high effort accounted for 63% of the total fix implementation effort.

5.2. Data mining approach and metrics for performance evaluation

To classify the level of the fix implementation effort we used three supervised learning methods.

- Naive Bayes (NB) uses Bayes formula to calculate the posterior probabilities and then assigns each input instance to the class value with the highest probability[4].
- J48 decision tree (Java implementation of C4.5) builds decision trees using the concept of information entropy, and

[4]Naive Bayes assumes independence among input features, which does not hold for our input features. In some cases, despite the unrealistic assumptions, Naive Bayes has been shown to work well in practice (e.g. [32]), which motivated us to include it in the experiments.

Table 8: Features used for prediction of the level of fix implementation effort

Fault type	Root cause of the failure. Categorical feature with values: requirements faults, design faults, coding faults, integration faults, and other faults.
Detection activity	Activity taking place when the failure was observed. Pre-release detection activities include inspection/audits, analysis, testing, and other. Detection can also be post-release.
Severity	Potential impact of the failure. Ordinal feature with values: safety-critical, non-critical, and unclassified.
Failed component	Component that the non-conformance SCR was associated with. Identifies one of the 21 components.
Release	Release of the failed component, with values from 1 - 7. (Each component is on its own release schedule.)
Investigation effort	Number of hours required to investigate and report the failure (i.e., non-conformance SCR).
Architectural tier	The system has a three-tier hierarchical architecture. This feature identifies the tier 1-3 to which the failed component belongs.
Architectural group	Architectural groups to which the failed component belongs, with values 1-7. Included because our previous work showed that fixes tend to remain within the architectural groups of the failed component [5].

- PART is a rule induction method based on partial decision trees.

As a baseline we used the ZeroR learner, which always predicts for the majority class. To be useful, other learners should perform better than ZeroR.

The imbalanced distribution of SCRs across the three effort levels is challenging for the learners. To treat the imbalanced data problem, we applied the following three sampling techniques with a goal to improve the balance of the dataset:

- *Oversampling* was done by resampling from the minority class (i.e., the high effort class) to increase the number of samples from 93 to 399, same as the number of instances in the low effort class.
- *Synthetic Minority Over-sample Technique (SMOTE)* creates new synthetic instances of the minority class using the nearest neighbor technique, rather than resampling existing instances. As for oversampling, SMOTE was used to increase the number of high effort instances to 399.
- *Undersampling* subsamples from the more dominant class(es). It was applied to force uniform distribution across the three classes (i.e, 93 instances in each effort category).

We used the Weka implementations of the described learners and imbalance treatments (i.e., sampling techniques) [33].

To evaluate the predictive abilities of all combinations of learners and imbalance treatments, we ran 10 fold cross validation. (For the 10 fold cross validation, the data is divided into 10 folds; the learner is trained on nine folds and the 10-th fold is used for testing. This process is repeated ten times, each time testing the learner on a different fold.)

The performance was compared using two approaches: (1) *per class performance* that evaluates the predictions for each class (i.e., high, medium, and low), and (2) *overall performance*.

Per class performance. We first compute the confusion matrix given by (1). For each class i (i.e. high, medium, low fix implementation effort), the element on the diagonal, E_{ii}, is equal to the number of SCRs that were correctly assigned to this class, which is referred to as the true positives TP_i. All other elements $E_{ij}, i \neq j$ give the number of instance that belong to the class i, but have been *incorrectly* assigned to class j.

$$\begin{bmatrix} E_{11} & E_{12} & E_{13} \\ E_{21} & E_{22} & E_{23} \\ E_{31} & E_{32} & E_{33} \end{bmatrix} \tag{1}$$

For each individual class i let:

- TP_i denote true positives - the number of SCRs that belong to i-th class and are correctly assigned to this class;
- FP_i denote false positives - the number of SCRs that *do not* belong to i-th class, but are *incorrectly* assigned to this class;
- FN_i denote false negatives - the number of SCRs that belong to i-th class, but are *falsely* not assigned to this class;
- TN_i denote true negatives - the number of SCRs that *do not* belong to i-th class and are correctly not assigned to this class.

Then, for each class we compute the *recall*, R_i, often referred to as *probability of detection*, which is defined as the ratio of SCRs correctly classified as class i to all SCRs in class i (equation 2), and *precision*, P_i, which determines the fraction of SCRs correctly classified to belong to class i out of all SCRs classified as class i (equation 3).

$$R_i = \text{TP}_i/(\text{TP}_i + \text{FN}_i) \tag{2}$$

$$P_i = \text{TP}_i/(\text{TP}_i + \text{FP}_i) \tag{3}$$

Finally, the *F-score* (F_i) of each class i is computed as a harmonic mean of R_i and P_i:

$$\text{F}_i = 2R_iP_i/(R_i + P_i) \tag{4}$$

The harmonic mean is appropriate when dealing with rates or ratios, as in case of recall and precision because it is not very sensitive to outliers and $\min(R_i, P_i) \leq F_i \leq \max(R_i, P_i)$. The F-score values are in the range $[0, 1]$, with larger values corresponding to better classification. Ideally, we want both recall and precision to be 1, which leads to F-score equal to 1. If either one is 0, the F-score is 0 by definition.

Overall performance is evaluated using the *accuracy* given by

$$\text{accuracy} = \text{TP}/(\text{TP} + \text{FN}) = \sum_{i=1}^{m} \text{TP}_i / \sum_{i=1}^{m} (\text{TP}_i + \text{FN}_i). \tag{5}$$

5.3. Prediction Results

Per class results for each learner for the original data and each of the imbalance treatments are presented in the form of heat maps in Figure 5, while R_i, P_i, and F_i values are given Table 9.

Based on the heat maps shown at the top of Figure 5 for 'No treatment' and associated metrics (i.e., R_i, P_i, F_i) shown in Table 9, we made the following observations for the learners' per class performance on the original dataset:

- **High fix implementation effort SCRs are the most difficult to classify.** NB did slightly better than J48 and PART, but correctly classified only 38% of all SCRs that required high fix implementation effort. NB also showed the highest F_i values.
- **Medium fix implementation effort SCRs are the easiest to classify correctly.** J48 and PART each correctly classified over 80% of SCRs that required medium fix implementation effort (i.e., 88% and 83%, respectively). NB correctly classified 70% of the the medium fix implementation SCRs. All learners showed similar P_i values (i.e., 72%). J48 had the highest and NB had the lowest F_i score (i.e., 79% and 71%, respectively).
- **Low fix implementation effort SCRs are classified fairly well.** NB performed the best for correctly classifying low effort SCRs (i.e., 67%), while J48 and PART correctly classified 58% and 62%, respectively. All learners had similar F_i scores (i.e., 65 - 68%).

Based on Figure 5 and Table 9, we made the following observations related to the imbalance treatments:

- **Using *Oversampling* or *SMOTE* significantly increased the prediction capabilities for the high effort SCRs.** This is important because high effort SCRs have significant impact on resource planning, and therefore their more accurate predictions are beneficial to practitioners. Specifically, the average recall of the high effort class for J48 and PART increased to above 84%, from 31% and 24%, respectively. This came at small cost of decreasing the recall for medium effort SCRs (in the range of 3% to 12%).

		ZeroR			NB			J48			Part		
		High	Med	Low	High	Med	Low	High	Med	Low	High	Med	Low
No Treatment	High	0%	100%	0%	38%	60%	2%	31%	68%	1%	24%	74%	2%
	Med	0%	100%	0%	5%	70%	24%	3%	88%	9%	4%	83%	12%
	Low	0%	100%	0%	1%	31%	67%	0%	42%	58%	1%	36%	62%
Oversampling	High	0%	100%	0%	64%	32%	4%	84%	15%	1%	86%	13%	0%
	Med	0%	100%	0%	12%	65%	24%	12%	76%	11%	13%	75%	12%
	Low	0%	100%	0%	3%	30%	67%	1%	32%	66%	2%	34%	64%
SMOTE	High	0%	100%	0%	67%	32%	1%	85%	15%	0%	84%	15%	1%
	Med	0%	100%	0%	18%	57%	25%	11%	78%	11%	8%	80%	13%
	Low	0%	100%	0%	5%	28%	67%	2%	34%	64%	2%	33%	66%
Undersampling	High	68%	32%	0%	66%	31%	3%	68%	25%	8%	68%	27%	5%
	Med	71%	29%	0%	23%	47%	30%	26%	53%	22%	23%	53%	25%
	Low	71%	29%	0%	15%	17%	68%	5%	19%	75%	6%	24%	70%

Figure 5: Heat maps for prediction of fix implementation effort using ZeroR, NB, J48, PART for the original data and each of the imbalance treatments applied. Rows represent the actual class (i.e., high, medium and low) and columns represent the predicted class value. The density of each cell, represented by shades of grey, gives the percentage of class instances assigned to a predicted class by the classifier. For each classifier (i.e., combination of learner and imbalance treatment), the main diagonal corresponds to correct classification assignments (R_i). For a good classifier, the cells on the main diagonal are the darkest among all cells.

Table 9: Recall (R_i), precision (P_i), and F-score (F_i) percentages for each class $i = high, medium, low$ and all combinations of learners and treatments

Imbalance treatment	Effort level	ZeroR			NB			J48			PART		
		R_i	P_i	F_i	R_i	P_i	F_i	R_i	P_i	F_i	R_i	P_i	F_i
No treatment	High	0	0	0	38	47	42	31	55	40	24	39	30
	Med	100	57	73	70	72	71	88	72	79	83	72	77
	Low	0	0	0	67	62	65	58	80	67	62	75	68
Oversampling	High	0	0	0	64	72	68	84	78	81	86	78	82
	Med	100	46	63	65	64	64	76	73	75	75	73	74
	Low	0	0	0	67	61	64	66	77	72	64	76	69
SMOTE	High	0	0	0	67	64	66	85	80	83	84	85	84
	Med	100	46	63	57	62	59	78	73	75	80	74	77
	Low	0	0	0	67	62	64	64	77	70	66	75	70
Undersampling	High	68	32	44	66	64	65	68	68	68	68	70	69
	Med	29	32	31	47	49	48	53	54	54	53	51	52
	Low	0	0	0	68	67	67	75	72	74	70	70	70

- **Using *Undersampling* led to a smaller improvement of the high effort recall at the expense of significantly worse medium effort recall values** when compared to oversampling and SMOTE.
- **All imbalance treatments led to some improvement of the low effort class recall** when compared to the original imbalanced dataset.

To compare the overall performance of the combinations of learners and imbalance treatments, we provide Table 10, which shows the mean overall accuracy and Figure 6, which contains the box plots of the overall accuracy produced using the results from the 10 fold cross validation. From these results, we made the following observations:

- **All learners did significantly better than the baseline learner ZeroR**, which always predicts for the majority class.
- **Decision tree learners, J48 and PART, outperformed the NB learner for the original dataset and all imbalance treatments**. Specifically, for the original data set J48 had the highest accuracy (73%), PART was slightly less accurate (71%), and NB had the lowest accuracy (67%).

Table 10: Mean accuracy values for predicting the level of fix implementation effort

Imbalance treatment	ZeroR	NB	J48	PART
No treatment	57%	67%	73%	71%
Oversampling	46%	65%	76%	75%
SMOTE	46%	62%	76%	77%
Undersampling	32%	60%	65%	63%

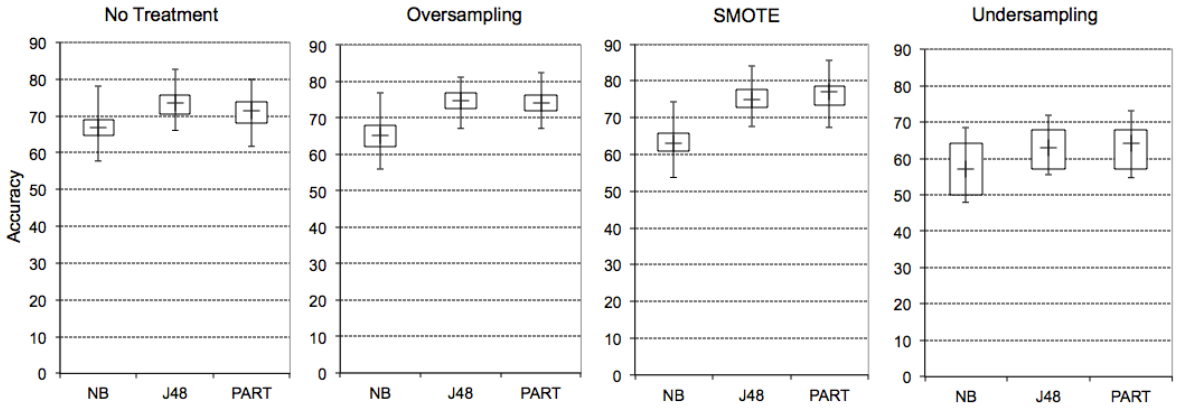

Figure 6: Box plots showing the accuracy values based on the 10 fold cross validation experiments

- **The overall accuracy of J48 and PART improved when using *Oversampling* and *SMOTE* compared to the original datset.** This is expected having in mind the per class results. The increased accuracy was due to significantly better predictions of the high effort SCRs at the cost of only a slight decrease in the ability to predict medium effort SCRs.

- Unlike *Oversampling* and *SMOTE*, ***Undersampling*** **decreased of the overall accuracy of J48 and PART, compared to the original dataset.**

- **None of the imbalance treatments improved the overall accuracy of NB learner.**

In general, it is difficult to compare prediction results with related works because different papers use different features, different prediction methods (i.e., prediction of numerical values versus classification), and different metrics to evaluate the performance. This task is even harder in case of fix implementation effort due to the following reasons: (1) many papers were focused on fixing individual faults as opposed to fixing all faults associated with individual failures and (2) most of the papers were focused on predicting the time to fix a fault measured in calendar days the bug report remained open, which is a very different response variable than the effort measured in actual hours spent fixing faults.

Of related works on prediction, the closest to ours are two papers [13, 20], which also used several levels of effort. However, [13] used log linear functional form for prediction and Kendalls τ for evaluation of the learner's performance, which makes the comparison hard. Song et al. [20] reported only accuracy, which for the Naive Bayes, C 4.5, and PART learners was similar to our results (i.e., from 68 - 72%), while for the association rules method was better. Note that none of the related works reported the per class performance metrics (i.e., recall, precision and F-score for each class) nor they applied treatments to address the class imbalanced problem which likely was present in other effort datasets as well.

6. Threats to validity

The threats to validity are grouped into the four categories: construct validity, internal validity, conclusion validity, and external validity.

Construct validity is concerned with whether we are measuring what we intend to measure. One threat to construct validity is translating data values to metrics without having sufficiently defined constructs. For example, the use

of inconsistent and/or imprecise terminology in the area of software quality assurance is a threat to validity that often makes comparisons across studies difficult. To address this threat, we provided the definitions of faults, failures, and fixes, as well as the definition of all attributes. Data values and their interpretation can also present a threat. To address this, we worked closely with project analysts to ensure we thoroughly understood the data and we were not misinterpreting any values. Another treat to construct validity is mono-operation bias, which is related to under-representation of the cause-construct. In other words, empirical studies and specifically case studies, often lack some type of data that, if available, could improve the interpretation of the results and/or help explaining the cause-effect relationships. In our case study, each non-conformance SCR (i.e., failure) was linked to one or more CN, which allowed as to connect the dots from the failure to faults that caused it and then to changes made to fix these faults. This allowed us to study the total effort for all changes made in association with an individual failure.

Internal validity threats are concerned with unknown influences that may affect independent variables and their interaction with dependent variables. Data quality is one of the major concerns to the internal validity. To ensure the data quality, we excluded SCRs that were missing mandatory fields, were duplicates, related to operator errors, and/or were withdrawn. Validity of effort data may be questionable depending on how they are measured and recorded. For example, in some studies effort data were deduced based on the dates when the issues were opened and closed. Work focused on development effort [28] has shown that in a working environment that is not controlled and subjects are not directly overseen, self-reported development effort values may be overestimated or underestimated, depending on the subject. For this NASA mission, analysts and developers recorded the investigation effort and fix implementation effort at a low granularity (in hours). Even though these effort values were self-reported, we believe that the data are of high quality because the work was overseen and the entered values were approved by the supervisors. Severity values may also be questionable and often are inflated because it is a common practice to prioritize fixing faults based on severity. To ensure the accuracy of severity values, in addition to values entered in the SCRs, we also examined the severity values assigned to the associated CNs. This allowed us to verify the severity values extracted from SCRs or, based on the recommendation made by the project analysts, to update them to the more accurate value given in the CNs. Finally, it is important to note that this mission has a review board which follows well-defined process and procedures for tracking SCRs and verifying or correcting the attribute values. This fact provides further assurance of the data quality.

Conclusion validity threats impact the ability to draw correct conclusions. One threat to conclusion validity is related to data sample sizes. The work presented in this paper is based on a large dataset consisting of fixes associated with over 1,200 failures linked to 2,500 faults. As any high-quality operational safety-critical system, the sample of post-release failures and the sample of safety-critical failures were both relatively small. Despite the fact that our dataset spans over 10-year period, only 3.2% of the non-conformance SCRs were reported on-orbit and only 10.6% where classified as safety-critical. Another threat to conclusion validity is related to using inappropriate statistical tests. We chose the statistical tests based on the validity of the underlying assumptions of the tests. That is, we used the appropriate non-parametric tests because our data were not normally distributed. Finally, it is important to mention that our work was done in close collaboration with NASA personnel and often involved multiple cycles of analyzing, presenting, and reviewing the data and interpretations. Their input and regular feedback most definitely contributed to the quality of the research and ensured accurate interpretation of the results.

External validity is concerned with the ability to generalize results. The breadth of this study, including the facts that it is (1) based on large NASA mission containing over two millions lines of code in over 8,000 files and (2) the system was developed at several locations by different teams over approximately 10 years of development and operation, allow for some degree of external validation. Nevertheless, we cannot claim that the results would be valid across all software products. It is widely accepted that generalizations are usually based on multiple empirical studies that have replicated the same phenomenon under different conditions [25, 34, 35]. Therefore, whenever possible we compared our results with related works, seeking to identify phenomena that are invariant across multiple domains. However, for the analyses that were done for the first time in this paper, the external validity remains to be established by future similar studies that will use other software products as case studies.

7. Conclusion

This paper focuses on exploring the investigation effort needed to report software change requests that were due to non-conformance with requirements (i.e., software failure) and the effort needed to fix all faults associated with a

particular failure. While prior work exists on prediction of fix implementation effort (i.e., fault correction effort), few papers have analyzed the effort and how it is affected by different factors. This paper addressed both the analysis and prediction of the fix implementation effort. The most interesting empirical findings from our study are summarized in Table 11 and compared with relevant results from related studies, if they existed.

The findings from this study were beneficial to the NASA mission, and potentially could benefit other large scale safety-critical systems. With respect to the mission, we worked closely with the NASA personnel and provided them with detailed results of our analysis. Even though some of our results confirmed the anecdotal knowledge of the mission's verification and validation team, the provided insights and quantification added value for sustained engineering of the mission. Other results of our analysis revealed patterns that were unknown and, to some extend, surprising to the mission team. These included the finding that fixes spread across artifacts were more costly than fixes spread across components and the finding that not all faults associated with later life-cycle required significant effort.

The findings of this study may benefit other systems that have similarities with the NASA mission. We believe that mining the information related to investigating and fixing software faults is helpful for building a knowledge base that can be reused on other, similar systems (e.g., safety-critical systems, systems that operate in partially understood operating environments, and systems that require high degree of autonomy). Exploring the same research questions on other case studies would test the generalizability of our findings and help establishing a set of findings that are invariant across different software systems. We hope that the software engineering researchers and practitioners will be encouraged to further improve the prediction models and use them to make predictions as new non-conformance software change requests are submitted, which will allow them to more effectively plan debugging and maintenance resources.

We conclude the paper with voicing the need of better record keeping of change and effort data. Improving the current bug tracking systems, which often do not establish links to changes made to fix the bugs and very rarely keep track of the effort spent on fixing the bugs, will provide basis for more widespread analysis and prediction of fix implementation effort and consequently affect the software development and testing practices.

Appendix: Comments on statistical tests and multiple comparison procedures

Software metrics, including effort data, typically follow skewed distributions. Therefore, the nonparametric techniques for hypotheses testing are better suited to the software engineering data than parametric tests. Nonparametric tests are often called "distribution-free" because they do not assume that data under analysis were drawn from a population distributed in a certain way, e.g., normally distributed. Other advantages of nonparametric tests include the facts that they can be used with ordinal scale (i.e., data expressed in ranks), are useful for small samples, and are not sensitive to outliers. Nonparametric tests, however, have lower statistical power than their parametric counterparts.

In RQ2 we explored if investigation effort and fix implementation effort associated with post-release failures are higher than the corresponding efforts associated with pre-release failures. The appropriate nonparametric test for this case (i.e., two independent samples consisting of pre-release and post-release failures) is the *Mann-Whitney test*, which is also called *Wilcoxon-Mann-Whitney test* or only *Wilcoxon test*. (Mann, Whitney, and Wilcoxon independently proposed nonparametric tests which are essentially the same [36].) The null hypothesis H_0 is that investigation efforts associated with pre-release failures and post-release failures have the same distribution. H_0 is tested against the directional hypothesis H_1 that the investigation effort associated with post-release failures is stochastically larger than the investigation effort associated with pre-release failures. (Note that another way of stating the alternative hypothesis, used by some authors, is that the median investigation effort associated with post-release failures is greater than the median investigation effort associated with pre-release failures.) The null and alternative hypotheses for the fix implementation effort are stated analogously.

In RQ3 we explored the amounts of investigation effort, and subsequently fix implementation effort, associated with three different levels of failures' severity: safety-critical, non-critical, and unclassified. A typical nonparametric test for this case would be the *Kruskal-Wallis test*, which tests the null hypothesis that k groups (in our case $k = 3$) come from the same population, i.e., identical populations with the same median $H_0 : \theta_1 = \theta_2 = \theta_3$. The alternative hypothesis in case of Kruskal-Wallis test is that at least one pair of groups has different medians and may be written as $H_1 : \theta_i \neq \theta_j$ for some groups i and j. Instead of Kruskal-Wallis test we used the *Jonckheere-Terpstra test* [36],

Table 11: Summary of main findings and comparison with related studies

	Description	Findings	Related studies	Section
RQ1	Effort distribution	20% of failures accounted for 75% of total investigation effort.	Not explored by others.	4.1
		20% of failures were associated with 83% of total fix implementation effort.	Consistent with [12, 13, 19].	
		Failures associated with high investigation effort did not always require high fix implementation effort and vice versa.	Not explored by others.	
RQ2	Pre- vs. post-release failures	Post-release failures required 1.5 times more investigation effort and 3 times more fix implementation effort than pre-release failures.	Consistent with [11] for fix implementation effort.	4.2
RQ3	Safety-critical vs. non-critical failures	Safety-critical failures required 2 times more investigation effort and 3 times more fix implementation effort than non-critical failures.	Not explored by others.	4.3
RQ4	Spread of fixes	Fix implementation effort increased with the spread of faults across components. Specifically, 39% of the total fix implementation effort was spent on only 16% of failures that were associated with fixes in multiple components.	Consistent with [13].	4.4
		Fix implementation effort increased with the spread of faults across multiple types of software artifacts. In particular, 60% of the total fix implementation effort was spent on 28% of failures associated with fixes in more than one type of artifact.	Not explored by others.	
RQ5	Types of faults & artifacts fixed	Some types of faults associated with later life-cycle activities did not require significant effort, especially if only one type of artifact was fixed.	Not explored by others.	4.5
RQ6	Interaction among factors	Fixes spread across artifacts were more costly than fixes spread across components.	Not explored by others.	4.6
		While the fix implementation effort related to pre-release failures increased with the spread of faults across components and artifacts, fix implementation effort for post-release failures was not sensitive to the spread of fixes.	Not explored by others.	
RQ7	Prediction of fix implementation effort	The level of fix implementation effort was predicted with 73% accuracy for the original, imbalanced dataset.	Consistent with [20] for Naive Bayes and decision tree learners.	5.3
		Oversampling and SMOTE increased the overall accuracy to 75-77%, due to significant improvement of the high effort level predictions and slight improvement of the low effort predictions.	Not explored by others.	

which allows us to test the above null hypothesis of "no difference" against an alternative hypothesis that the groups are ordered in a specific a priori sequence. This alternative hypothesis may be written as $H_1 : \theta_1 \leq \theta_2 \leq \theta_3$. If the alternative hypothesis is true, at least one of the differences is a strict inequality ($<$). In this case, even though the alternative hypothesis associated with the Kruskal-Wallis test is valid, it is too general. Since the severity levels are ordered, it made sense to test if the amounts of effort associated with these severity levels are also ordered. Note that both Kruskal-Wallis test and Jonckheere-Terpstra test account for multiple comparisons (i.e., are developed specifically for comparison of $k > 2$ groups) and do not require significance levels adjustment as in cases when one would use Mann-Whitney test to do multiple pairwise comparisons (i.e., three pairwise comparisons if $k = 3$).

Similarly as in RQ3, in RQ4 we used the Jonckheere-Terpstra test to test for ordered alternatives of amounts of effort associated with different levels of spread across components, as well as different levels of spread across software artifacts. Note that the order here was also chosen a priori, assuming that larger spread leads to more fix implementation effort.

Next, we address the issue of adjustments for multiple comparisons, which has been a focus of considerable debate, especially when it is related to analysis-wide comparisons [37]. Note that the significance level used for an individual test is the marginal probability of falsely rejecting the hypothesis, regardless of whether the remaining hypotheses are rejected. In some research fields, with a goal to reduce the type I error (i.e., false positives), in case of multiple comparisons, authors recommend adjusting the significance levels for individual tests to restrict the overall analysis-wide type I error to 0.05 [38]. Unfortunately, a byproduct of correcting for multiple comparisons is increasing the type II error, that is, the number of false negatives (when a true, existing pattern cannot be detected as statistically significant) [37]. The theoretical basis for advocating a routine adjustment for multiple comparisons is the "universal null hypothesis" that chance serves as the first-order explanation for observed phenomena. Some statisticians believe that "no empiricist could comfortably presume that randomness underlies the variability of all observations" and recommend never correcting for multiple comparisons while analyzing data [39]. Instead one should report all of the individual p values and make it clear that no mathematical correction was made for multiple comparisons.

In general, there is an inherent trade-off between protecting against type I errors (declaring differences statistically significant when they were in fact just due to chance) and type II errors (declaring a difference is not statistically significant even when there really is a difference). Therefore, a middle ground approach to testing multiple hypotheses is as follows [40]. In cases when a researcher wants to test a theory with multiple implications that must simultaneously hold for the theory to survive the test, then the failure of a single implication (as an independent hypothesis) defeats the theory. In this case, a composite hypothesis test (also called a family of hypotheses), with adjustments for multiple comparisons is indicated. (For example, if testing the abovementioned hypothesis $H_0 : \theta_1 = \theta_2 = \theta_3$ is done by using pairwise tests (e.g., Mann-Whitney test), adjusting for multiple comparisons is necessary.) On the other side, if a conclusion would follow from a single hypothesis, fully developed, tested, and reported in isolation from other hypotheses, then a single hypothesis test is warranted.

In the case of the analyses presented in section 4, we do not build a theory that requires all individual tests to hold simultaneously, and therefore there is no need to adjust the significance level for multiple comparisons. Rather, we test a total of six individual hypothesis.

If of interest, one may create a composite hypothesis, for example in case of RQ2 that post-release failures have both higher investigation effort and higher fix implementation effort (i.e., both these implications must hold for the composite hypothesis to hold). The simplest adjustment for multiple comparisons is to use the Bonferroni procedure which controls the family-wise error rate [41]. Thus, if one is doing n statistical tests, the critical value for an individual test would be $\alpha^* = \alpha/n$, and one would only consider individual tests with $p < \alpha^*$ to be significant. In case of the composite hypothesis for RQ2, each individual test has to be significant at $\alpha^* = 0.05/2 = 0.025$ for the composite hypothesis to be significant at $\alpha = 0.05$. Having in mind that both p-values are smaller than 0.025 (see section 4.2), we can conclude that post-release failures have both higher investigation effort and higher fix implementation effort and the result is statistically significant at $\alpha = 0.05$ level. Similarly, one may create composite hypothesis for each of the research questions RQ3 and RQ4, which will also be statistically significant based on the Bonferroni procedure.

Since pre-release / post-release failures (RQ2), the severity levels (RQ3), and the spread of fixes (RQ4) are different dimensions, we believe that there is no need to test a composite hypothesis consisting of all six individual tests.

Acknowledgments

This work was funded in part by the NASA OSMA Software Assurance Research Program under a grant managed through the NASA IV&V Facility. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency and the NASA personnel. The authors thank the following NASA personnel for their invaluable support: Jill Broadwater, Pete Cerna, Randolph Copeland, Susan Creasy, James Dalton, Bryan Fritch, Nick Guerra, John Hinkle, Lynda Kelsoe, Gary Lovstuen, Tom Macaulay, Debbie Miele, Lisa Montgomery, James Moon, Don Ohi, Chad Pokryzwa, David Pruett, Timothy Plew, Scott Radabaugh, David Soto, and Sarma Susarla.

References

[1] Cambridge University report on cost of software faults, http://www.prweb.com/releases/2013/1/prweb10298185.htm (2013).
[2] M. Griss, Software reuse: From library to factory, IBM Systems Journal 32 (1993) 548–565.
[3] B. Boehm, V. Basili, Software defect reduction top 10 list, IEEE Computer 34 (1) (2001) 135–137.
[4] ISO/IEC/IEEE 24765 Systems and Software Engineering Vocabulary, Geneva, Switzerland (2010).
[5] M. Hamill, K. Goseva-Popstojanova, Exploring the missing link: an empirical study of software fixes, Software Testing Verification and Reliability 24 (5) (2013) 49–71.
[6] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for Eclipse, 3rd International Workshop on Predictor Models in Software Engineering (2007) 9.
[7] M. Hamill, K. Goseva-Popstojanova, Common trends in fault and failure data, IEEE Transactions on Software Engineering 35 (4) (2009) 484–496.
[8] M. Hamill, K. Goseva Popstojanova, Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system, Software Quality Journal 23 (2) (2015) 229–265.
[9] M. Jorgensen, A review of studies of expert estimation of software development effort, Journal of System and Software 70 (1-2) (2004) 37–64.
[10] H. Siy, A. Mockus, Measuring domain engineering effects on software change cost, in: 6th IEEE International Software Metrics Symposium, 1999, pp. 304–311.
[11] A. Mockus, D. Weiss, P. Zhang, Understanding and predicting effort in software projects, in: International Conference of Software Engineering, 2002, pp. 274–284.
[12] A. Ahsan, J. Ferzund, F. Wotawa, Program file bug fix effort estimation using machine learning methods for open source software projects, Tech. rep., Institute for Software Technology (April 2009).
[13] W. Evanco, Prediction models for software fault correction effort, in: 5th European Conference on Software Maintenance and Reengineering, Libson, Portugal, 2001, pp. 114–120.
[14] H. Zeng, D.Rine, Estimation of software defects fix effort using neural networks, in: 28th International Computer Software and Applications Conference, 2004, pp. 20–21.
[15] L. Paner, Predicting Eclipse bug lifetimes, in: 4th International Workshop Mining Software Repositories, 2007, pp. 29–.
[16] C. Weiss, R. Premraj, T. Zimmerman, A. Zellar, How long will it take to fix this bug?, in: 4th International Workshop on Mining Software Repositories, 2007, pp. 1–.
[17] E. Giger, M. Pinzger, H. Gall, Predicting the fix time of bugs, in: Recommendation Systems for Software Engineering, 2010, pp. 52–56.
[18] W. Abdelmoez, M. Kholief, F. Elsalmy, Bug fix-time prediction model using naive bayes classifier, in: 22nd International Conference on Computer Theory and Application, 2012, pp. 167–172.
[19] H. Zhang, L. Gong, S. Versteeg, Predicting bug-fixing time: An empirical study of commerical software projects, in: International Conference on Software Engineering, 2013, pp. 1042–1051.
[20] Q. Song, M. Shepperd, M. Cartwright, C. Mair, Software defect association mining and defect correction effort prediction, IEEE Transaction on Software Engineering 32 (2) (2006) 69–82.
[21] E. Arishold, L. Briand, E. Johannssen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, The Journal of Systems and Software 83 (2009) 2–17.
[22] P. Runeson, M. Host, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2009) 131–164.
[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer-Verlag Berlin Heidelberg, 2012.
[24] C. Robson, Real world Reseach, Blackwell, Ontario, 2002.
[25] R. K. Yin, Case Study Research: Design and Methods, SAGE Publication Ltd, Thousand Oaks, CA, 2014.
[26] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. Emam, J. Rosenberg, Preliminary guidelines for empirical research in software engineering, IEEE Transaction on Software Engineering 28 (8) (2002) 721–734.
[27] K. Petersen, C. Wohlin, Context in industrial software engineering research, 3rd International Symposium on Empirical Software Engineering and Measurement (2009) 401–404.
[28] L. Hochstein, V. R. Basili, M. V. Zelkowitz, J. K. Hollingsworth, J. Carver, Combining self-reported and automatic data to improve programming effort measurement, in: 10th European Software Engineering Conference, 2005, pp. 356–365.
[29] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Wong, Orthogonal defect classification – a concept for in-process measurement., IEEE Transactions on Software Engineering 18 (11) (1992) 943–956.

[30] W. Conover, Practical Nonparametric Statistics, 3rd Edition, Wiley, 1999.
[31] B. Boehm, C. Abts, A. Brown, S. Chulani, B. Clark, E. Horowitx, R. Madachy, D. Reifer, D. Steece, Software Cost Estimation with Cocomo II, Prentice Hall, 2000.
[32] A. Garg, D. Roth, Understanding probabilistic classifiers, in: 12th European Conference on Machine Learning, 2001, pp. 179–191.
[33] Weka collection of machine learning algorithms for data mining tasks, `http://www.cs.waikato.ac.nz/ml/weka`.
[34] C. Andersson, P. Runeson, A replicated quantitative analysis of fault distributions in complex software systems, IEEE Transactions of Software Engineering 33 (5) (2007) 273–286.
[35] B. Kitchenham, The role of replications in empirical software engineering–a word of warning, Empirical Software Engineering 13 (2) (2008) 219–221.
[36] S. Siegel, N. J. C. Jr., Nonparametric Statistics for The Behavioral Sciences, McGraw-Hill, Inc., 1988.
[37] J. H. McDonald, Handbook of Biological Statistics, Sparky House Publishing, 2014.
[38] S. E. Maxwell, H. D. Delaney, Designing Experiments and Analyzing Data: A Model Comparison Perspective, Lawrence Erlbaum Associates, 2000.
[39] K. J. Rothman, No adjustments are needed for multiple comparisons, Epidemiology 1 (1) (1990) 43–46.
[40] P. J. Veazie, When to combine hypotheses and adjust for multiple tests, Health Services Research 41 (3 Pt 1) (2006) 804–818.
[41] Y. Hochberg, A. C. Tamhane, Multiple Comparison Procedures, Wiley Series in Probability and Statistics, 1987.